Type Classes in Functional Logic Programming

Author's version for E-Prints Complutense

Enrique Martin-Martin
Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Madrid, Spain
emartinm@fdi.ucm.es

Abstract

Type classes provide a clean, modular and elegant way of writing overloaded functions. Functional logic programming languages (FLP in short) like Toy or Curry have adopted the Damas-Milner type system, so it seems natural to adopt also type classes in FLP. However, type classes has been barely introduced in FLP. A reason for this lack of success is that the usual translation of type classes using *dictionaries* presents some problems in FLP like the absence of expected answers due to a bad interaction of dictionaries with the call-time choice semantics for non-determinism adopted in FLP systems.

In this paper we present a *type-passing* translation of type classes based on *type-indexed functions* and *type witnesses* that is well-typed with respect to a new liberal type system recently proposed for FLP. We argue the suitability of this translation for FLP because it improves the dictionary-based one in three aspects. First, it obtains programs which run as fast or faster—with an speedup from 1.05 to 2.30 in our experiments. Second, it solves the mentioned problem of missing answers. Finally, the proposed translation generates shorter and simpler programs.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Polymorphism; D.3.2 [*Language Classifications*]: Multiparadigm languages

General Terms Languages, Design, Performance.

Keywords Type Classes, Functional Logic Programming, Type-indexed functions.

1. Introduction

Type classes [10, 30] are one of the most successful features in Haskell. They provide an easy syntax to define overloaded functions—*classes*—and the implementation of those functions for different types—*instances*. Type classes are usually implemented by means of a source-to-source transformation that introduces extra parameters—called *dictionaries*—to overloaded functions [10, 30], generating Damas-Milner [7] correct programs. Dictionaries are data structures containing the implementation of overloaded functions for specific types and dictionaries for the superclasses. The efficiency of translated programs—using several optimizations [4, 11]—and the fact that the translation handles correctly multiple modules and separate compilation, have resulted in that nowadays it is the most used technique for implementing type classes in functional programming (FP). Another scheme for translating type classes is passing type information as extra arguments to overloaded functions [29]. In this scheme, overloaded functions use a *typecase* construction in order to pattern-match types and decide which concrete behavior—*instance*—to use. Although it is possible to encode it using *generalized algebraic data types* (GADTs) [6, 14] or Guarded Recursive Datatype Constructors [31], this translation scheme has not succeeded in the FP community.

Functional logic programming (FLP) [12] aims to combine the best of declarative paradigms (functional, logic and constraint languages) in a single model. FLP languages like Toy [22] or Curry [13] have a strong resemblance to lazy functional languages like Haskell [15]. However, a remarkable difference is that functional logic programs can be non-confluent, giving raise to so-called *non-deterministic functions*, for which a *call-time choice* semantics [8] is adopted. The following program is a simple example, using Peano natural numbers given by the constructors z and s[1]: *coin* $\rightarrow z$, *coin* $\rightarrow s\ z$, *dup* $X \rightarrow$ *pair* $X\ X$—where *pair* is the constructor symbol for pairs. Here, *coin* is a non-deterministic function (*coin* evaluates to z and $s\ z$) and, according to call-time choice, *dup coin* evaluates to *pair* $z\ z$ and *pair* $(s\ z)\ (s\ z)$ but not to *pair* $z\ (s\ z)$ or *pair* $(s\ z)\ z$. Operationally, call-time choice means that all copies of a non-deterministic subexpression (*coin* in the example) created during reduction share the same value.

Functional logic languages have adopted the Damas-Milner type system, although it presents some problems when applied directly [9, 21]. However, with the exception of some preliminary proposals as [26]—presenting some ideas about type classes and FLP not further developed—and [23]—showing some problems that the dictionary approach produces when applied to FLP systems—type classes have not been incorporated in FLP. From the point of view of the systems, only an experimental branch of [1] and the experimental systems [2, 3] have tried to adopt type classes. One reason for this limited success is the problems presented in [23]. In addition to them, another important issue to address is the lack of expected answers when combining non-determinism and *nullary*[2] overloaded functions [24]. This problem is shown in the program in Fig. 1, taken from [24]. We use a syntax of type classes and instances similar to Haskell but following the mentioned syntactic convention adopted in the Toy system. The program contains an overloaded function `arb` which is a non-deterministic generator, and its instance for booleans. It also contains a function `arbL2` which returns a list of two elements of the

http://doi.acm.org/10.1145/1929501.1929524.

[1] We follow the syntactic conventions of Toy where identifiers are lowercased and variables are uppercased.

[2] i.e. of arity 0.

```
class arb A where
  arb :: A

instance arb bool where
  arb → false
  arb → true

arbL2 :: arb A => list A
arbL2 → [arb, arb]
```

a) Original program

```
data dictArb A = dictArb A

arb :: dictArb A -> A
arb (dictArb F) → F

arbBool :: bool
arbBool → false
arbBool → true

dictArbBool :: dictArb bool
dictArbBool → dictArb arbBool

arbL2 :: dictArb A -> list A
arbL2 DA → [arb DA, arb DA]
```

b) Translated program using dictionaries

Figure 1. Program containing a type class with a constant non-deterministic overloaded function

same instance of `arb`. Fig. 1-b) contains the translated program following the standard translation using dictionaries [10, 30]. The `arb` type class generates a data declaration for `arb` dictionaries—`dictArb`—and a projecting function `arb` to extract the concrete implementation from the dictionary. The instance `arb bool` generates a concrete dictionary—`dictArbBool`—and the `arbL2` function is transformed to accept an `arb` dictionary as first argument and pass it to the `arb` functions in its right-hand side. Expected results for the expression `arbL2::(list bool)` are `[true, true]`, `[true, false]`, `[false, true]` and `[false, false]`, however its evaluation in the translated program only produces `[true, true]` and `[false, false]`. The reason is the call-time choice semantics. The translated expression `arbL2 dictArbBool` reduces to `[arb dictArbBool, arb dictArbBool]`, but both copies of `dictArbBool` must share their value. Therefore they cannot be reduced to `dictArb true` and `dictArb false` in the different occurrences of the right-hand side, losing two expected solutions.

In this paper we propose and evaluate a type-passing translation of type classes for FLP based on type-indexed functions—functions with a different behavior for different types [14]—and type witnesses—representations of types as data values—that is well-typed in a new liberal type system recently proposed for FLP [20]. The proposed translation is not integrated in the type checking phase as in [10, 30], but it is a separated phase after type checking. This previous type checking phase is assumed to use a standard type system supporting type classes [5, 27], and decorates the function symbols with the inferred types.

We show that the proposed translation is a suitable option for FLP compared to the classical dictionary-based translation because of three reasons. First, it obtains programs which run as fast or faster—with and speedup ranging from 1.05 to 2.30 in our experiments. When we apply optimizations to both translated programs the speedup still remains favorable to the proposed translation. Second, it solves the mentioned problem of missing answers when combining non-determinism and nullary overloaded functions. Finally, the proposed translation has a similar complexity to the dictionary-based one, but generates shorter and simpler programs.

The following list summarizes the main contributions of the paper and at the same time presents the structure of the paper.

- We formalize a type-passing translation for type classes in FLP in Sect. 3. Although the broad idea of using such kind of translation is not a novelty [29], its concrete realization and the application to FLP, relying in a new type system [20], are new. In particular, the liberality of the type system avoids the need of a *typecase* construction in the target language, resulting in that translated programs do not need to enhance the syntax of FLP systems with that construction.
- We have measured the execution time of a collection of different programs involving overloaded functions that can be part of bigger real FLP programs—see Sect 4.1. Some of these programs have been adapted from the *nobench* suite of benchmark programs for Haskell. The speedup results—from 1.05 to 2.30—show that when no optimizations are applied, programs translated using the proposed type-passing scheme perform faster than those translated using the dictionary-based translation.
- There are several well-known optimizations than can be applied to translated programs using the dictionary-based scheme [4, 11]. In Sect. 4.1 we present some optimizations to the proposed type-passing translation. We have repeated the execution time measurements to the optimized programs, and we have checked that the proposed translation still obtains faster programs even when optimizations are applied.
- We study how the proposed translation solves the problem of missing answers that appears when combining non-determinism and nullary overloaded functions—see Sect. 4.2.
- In Sect. 5 we discuss some additional aspects—including some problems—that arise with the translations of type classes in FLP.

2. Preliminaries

This section introduces the syntax of types, the source language and the target language of the proposed translation. It also introduces the liberal type system in which the translated programs are well-typed.

2.1 Syntax

Fig. 2 gives the syntax of types, which are the usual ones when using type classes [10]. The only difference is that class names can have a mark $^\bullet$. We use this mark in the translation to distinguish between which class constraints generate a type information to pass to overloaded functions, as we will explain in Sect. 3. Overloaded types are simple types enclosed with a *saturated* context. Notice that in a saturated context class restrictions not only affect type variables but they can affect simple types as $list\ bool$ or $pair\ int\ (list\ nat)$. Contexts, which express class constraints over type variables, will be used in class and instance declarations. Type schemes are the same as in the Damas-Milner type system [7], and play the usual role to handle *parametric* polymorphism.

The syntax of source programs of the translation is shown in Fig. 3. It is the usual syntax for programs with type classes of one argument [10] adapted to Toy's syntax. We assume a denumerable set of data variables (X), and a set of function symbols (f) and constructor symbols (c), all them with associated arity. We say that a function is a *member* of a type class if it is declared inside that type class declaration, and it is an *overloaded function* if its inferred type has class constraints in the context. Notice that member function are overloaded functions, since they have exactly one class constraint in the context of its type. Patterns—our notion of values—are a subset of expressions. Notice that con-

Type variable	$\alpha, \beta, \gamma \ldots$	
Type constructor	C	
Class name	κ , $\kappa^\bullet$	
Simple type	τ	$::= \alpha \mid \tau \to \tau'$
		$\mid C\ \overline{\tau_n}$ with n = arity(C), n ≥ 0
Context	θ	$::= \langle \overline{\kappa_n\ \alpha_n} \rangle$ with n ≥ 0
Saturated context	ϕ	$::= \langle \overline{\kappa_n\ \tau_n} \rangle$ with n ≥ 0
Overloaded type	ρ	$::= \phi \Rightarrow \tau$
Type scheme	σ	$::= \forall \overline{\alpha_n}.\tau$ with n ≥ 0

Figure 2. Syntax of types

function symbol	f	
constructor symbol	c	
data variable	X	
program	::=	$\overline{data}\ \overline{class}\ \overline{inst}\ \overline{type}\ \overline{rule}$
data	::=	`data` $C\ \overline{\alpha} = c_1\ \overline{\tau} \mid \ldots \mid c_k\ \overline{\tau}$
class	::=	`class` $\theta \Rightarrow \kappa\ \alpha$ `where` $\overline{f :: \tau}$
inst	::=	`instance` $\theta \Rightarrow \kappa\ (C\ \overline{\alpha})$ `where` $\overline{f\ \overline{t} \to e}$ with $\overline{t}$ linear
type	::=	$f :: \theta \Rightarrow \tau$
rule r	::=	$(f :: \rho)\ \overline{t} \to e$ with $\overline{t}$ linear
pattern t	::=	$X \mid c\ \overline{t_n}$ with n $\leq$ arity(c)
		$\mid f\ \overline{t_n}$ with n $<$ arity(f)
expression e	::=	$X \mid c \mid f :: \rho \mid e\ e \mid let\ X = e\ in\ e$

Figure 3. Syntax of source programs

structor and function symbols partially applied to patterns—called HO-patterns—are considered as patterns in our setting, the HO Constructor-based conditional ReWriting Logic (HO-CRWL) approach to FLP [25] followed by the Toy system. This corresponds to an intensional view of functions, i.e., different descriptions of the same 'extensional' function can be distinguished by the semantics. In program rules (r) the set of patterns $\overline{t}$ is linear (there is not repetition of variables) and there are not extra variables in the right-hand side. However we do not support HO-patterns made with overloaded function symbols in the left-hand side of rules, due to some complications that arise during translation—see Sect 5.3. A particularity of the syntax is that function symbols in rules and expressions are always decorated with an overloaded type. We assume that this decoration comes from a previous type checking phase, and reflects to which types are functions applied. In the type checking stage the type checker decorates function symbols with a variant of its type, and instantiate it with the proper type of the application. For example if `eq` has the usual type $\langle eq\ A \rangle \Rightarrow A \to A \to bool$, a rule for a function `g`:

`g X` $\to$ `eq X [true]`

will have the decoration

`g::`$\langle\rangle \Rightarrow (list\ bool) \to bool$ `X` $\to$
`eq::`$\langle eq\ (list\ bool)\rangle \Rightarrow (list\ bool) \to (list\ bool) \to bool$
`X [true]`

In the right-hand side of `g`, the saturated context $\langle eq\ (list\ bool)\rangle$ indicates that the overloaded `eq` function is applied to elements of type $list\ bool$, so it needs that type information. The function `g` in the left hand side does not have any context because its context is *reduced* during type checking—see Sect. 3.3—and became empty, so it does not appear in the inferred type for `g`.

The syntax of target programs is similar to source programs, except that there are not class or instance declarations, function symbols in rules and expressions are not decorated with type information and type declarations for functions are only simple types.

```
size :: A -> nat
size false → s z
size true → s z
size z → s z
size (s X) → s (size X)

eq :: A -> A -> bool
eq true true → true
eq false false → true
eq z z → true
eq (s X) (s Y) → eq X Y
```

Figure 4. Examples of type-indexed functions

2.2 Liberal type system for FLP

The type system considered for the target language is a new simple extension of the Damas-Milner type system recently proposed for FLP [20]. The typing rules for expressions correspond to the well-known variation of Damas-Milner type system [7] with syntax-directed rules. The type inference algorithm $\Vdash\!\!\vdash$ follows the same ideas that algorithm $\mathcal{W}$ [7], however we have given the type inference a relational style $\mathcal{A} \Vdash\!\!\vdash e : \tau|\pi$. This algorithm accepts a set of type scheme assumptions $\mathcal{A}$ over symbols s_i which can be variables or constructor/function symbols—$\{\overline{s_n : \sigma_n}\}$—and an expression e, returning a simple type τ and a type substitution π—$[\overline{\alpha_n/\tau_n}]$. Intuitively, τ is the "most general" type which can be given to e, and π the "most general" substitution we have to apply to $\mathcal{A}$ in order to be able to derive any type for e. The difference is that, unlike FP, we cannot write programs as expressions—we do not have λ-abstractions—so we need an explicit method for checking whether a program is well-typed. We will say that a program is well-typed wrt. a set of assumptions if all the rules are well-typed:

DEFINITION 1. *A rule $f\ \overline{t} \to e$ is well-typed wrt. to a set of assumptions $\mathcal{A}$ iff:*

- $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash\!\!\vdash f\ \overline{t} : \tau_L|\pi_L$
- $\mathcal{A} \oplus \{\overline{X_n : \beta_n}\} \Vdash\!\!\vdash e : \tau_R|\pi_R$
- $\exists\pi.(\tau_L, \overline{\alpha_n \pi_L}) = (\tau_R, \overline{\beta_n \pi_R})\pi$

where $\overline{X_n}$ are the variables in $\overline{t}$, $\oplus$ is the symbol for the usual union of sets of assumptions and $\overline{\alpha_n}$, $\overline{\beta_n}$ are fresh type variables.

Intuitively, a rule is well-typed if the types $(\tau_R, \overline{\beta_n \pi_R})$ inferred for the right-hand side and its variables are more general than the types $(\tau_L, \overline{\alpha_n \pi_L})$ inferred for its left-hand side and its variables. Notice that programmers must provide an explicit type for every function symbol, otherwise the first point of the definition fails to infer the type for the expression $f\ \overline{t}$. Therefore Def. 1 cannot be used to infer the types of the functions, but to check that the types provided for the functions are correct.

The most remarkable feature of this new system is its liberality, that allows the programmer to define type-indexed functions in a very easy way, but still assuring essential safety properties like *type preservation* and *progress*—see [20] for more details. Consider the type-indexed functions `size` and `eq` defined over natural and booleans that appear in Fig. 4. The first three rules for `size` are well-typed because the type inferred for the right-hand side (nat) is more general than the inferred in the left-hand side (nat again). In the fourth rule the types inferred for the left-hand side and the variable `X` are both nat, and in the right-hand side the inferred types are nat and β resp., so the rule is well typed since (nat, β) is more general than (nat, nat). The same happens in the fourth rule of `eq`, where $(bool, \beta, \beta)$ inferred for the right-hand side is more general than $(bool, nat, nat)$ inferred for the left-hand side. The rest of rules for `eq` are well-typed for similar reasons.

3. Translation

As we have said in Sect. 1, the translation follows a type-passing scheme [29] and uses type-indexed functions and type witnesses. Instead of passing dictionaries containing the concrete implementation of the overloaded functions to use, in this scheme we pass data values—type witnesses—representing the types to which overloaded functions are applied. In the source program, saturated contexts that decorate function symbols show what types are they applied to, so we use that information to generate the concrete type witnesses. Member functions are translated into type-indexed functions that pattern-match on the type witness and decide which instance of the overloaded function to use. Due to the liberality of the type system, these type-indexed functions are encoded with type witnesses without the need of a special *typecase* constructions as in [29], so translated programs are usual FL programs.

3.1 Type witnesses

Type witnesses are data values that represent types. In [6, 14] these *type representations* are encoded using a GADT containing all the type representations. We follow a slightly different approach: we extend every data declaration with a new constructor in order to represent the type of the declared data. For example, a data declaration for Peano naturals `data nat = z | s nat` is extended with the constructor `#nat`, resulting in `data nat = z | s nat | #nat`; and a data declaration for lists `data list A = nil | cons A` is extended to `data list A = nil | cons A | #list A`. This extension of data declarations can be easily performed by the system. An interesting point of type witnesses defined this way is that they have exactly the same type they represent. In the previous example, `#nat` has type nat, and `#list (#list #nat)` has type *list (list A)*. This link between types and type witnesses allows us to generate automatically the type witness of a given simple type, fact that is used during translation.

DEFINITION 2 (Generation of type witnesses).
- *testify*(α) = X_α
- *testify*($C\ \tau_1 \dots \tau_n$) = *#C testify*(τ_1) $\dots$ *testify*(τ_n)

The function testify returns the same data variable X_α for the same type variable α. Notice that the testify function is not defined for functional types $\tau \to \tau'$. This is because we consider a source language where instances over functional types are not possible, so in the translation we will not need to generate type witnesses for that types. However, in our liberal type system it would be simple to create type witnesses for those types using a special data constructor `#arrow` of type $\alpha \to \beta \to (\alpha \to \beta)$.

3.2 Translation

In the classical dictionary-based scheme [10, 30], the translation is integrated in the type checking phase so that it uses the inferred type information. In this paper we follow a different approach, supposing that the translation from type classes to type-indexed functions comes after a type checking phase that has inferred the types to the whole program [5, 27]. Since the inferred type information is needed for the translation, we assume that the type checking phase has decorated the function symbols with their corresponding types. The idea of the translation is simple: we inspect the context of the types that decorate function symbols and extract from them the concrete type witnesses that we need to pass to the functions. We define a set of translation functions for the different constructions (whole programs, data declarations, classes, instances, type declarations, rules and expressions):

DEFINITION 3 (Translation functions).

$$trans_{prog}(\overline{data}\ \overline{class}\ \overline{inst}\ \overline{type}\ \overline{rule}) =$$
$$\overline{trans_{data}(data)}\ \overline{trans_{class}(class)}\ \overline{trans_{inst}(inst)}$$
$$\overline{trans_{type}(type)}\ \overline{trans_{rule}(rule)}$$

$$trans_{data}(\texttt{data}\ C\ \overline{\alpha} = c_1\ \overline{\tau} \mid \dots \mid c_k\ \overline{\tau}) =$$
$$\texttt{data}\ C\ \overline{\alpha} = c_1\ \overline{\tau} \mid \dots \mid c_k\ \overline{\tau} \mid \#C\ \overline{\alpha}$$

$$trans_{class}(\texttt{class}\ \theta \Rightarrow \kappa\ \alpha\ \texttt{where}\ \overline{f :: \tau}) = \overline{f :: \alpha \to \tau}$$

$$trans_{inst}(\ \texttt{instance}\ \theta \Rightarrow \kappa\ (C\ \overline{\alpha})\ \texttt{where}\ \overline{f\ \overline{t} \to e}) =$$
$$\overline{f\ testify(C\ \overline{\alpha})\ \overline{trans_{expr}(t)} \to trans_{expr}(e)}$$

$$trans_{type}(f :: \theta \Rightarrow \tau) = f :: \alpha_1 \to \dots \to \alpha_n \to \tau$$
where $\alpha_1 \dots \alpha_n$ appear in θ constrained by a class marked with $^\bullet$

$$trans_{rule}((f :: \rho)\ \overline{t} \to e) =$$
$$trans_{expr}(f :: \rho)\ \overline{trans_{expr}(t)} \to trans_{expr}(e)$$

$$trans_{expr}(X) = X$$
$$trans_{expr}(c) = c$$
$$trans_{expr}(f :: \rho) = f\ testify(\tau_1) \dots testify(\tau_n)$$
where $\rho \equiv \phi \Rightarrow \tau$ and $\tau_1 \dots \tau_n$ appear in ϕ constrained by a class marked with $^\bullet$
$$trans_{expr}(e\ e') = trans_{expr}(e)\ trans_{expr}(e')$$
$$trans_{expr}(let\ X = e\ in\ e') =$$
$$let X = trans_{expr}(e)\ in\ trans_{expr}(e')$$

The translation of a program is simply the translation of its components. Data declarations are extended with the constructor of its type witness as explained in Sect. 3.1. Class declarations generate type declarations for the type-indexed functions. The generated type is the same as the one declared in the class but it has an extra first argument for the type witness. Consider the class declaration for the class `foo`:

```
class foo A where
  foo :: A → bool
```

This declaration generates a type declaration for the type-indexed function `foo` adding an extra first argument A to the type of the member function. This argument A is the type variable of the type class:

```
foo :: A → A → bool
```

Type declarations are treated in a similar way, with the difference that we only add new arguments to the translated type if they are constrained by a class with a $^\bullet$ mark, i.e., if the corresponding type witnesses are needed. Consider the type declaration for `f`:

```
f :: ⟨eq• A, ord A, eq• B⟩ ⇒ A → B → bool
```

This declaration generates a type declaration with the extra arguments A and B—and in that order—which are the type variables constrained by marked class names in the context:

```
f :: A → B → A → B → bool
```

Rules in an instance declaration are translated one by one. These rules generate the rules of type-indexed functions, so we add a type witness of the concrete instance as the first argument so they dispatch on it. Notice that a rule generated from an instance do not need any extra type-witness, since the type declared in the class declaration is a simple type and does not have a context. Consider the instance declaration `foo` for *list* A:

```
instance foo (list A) where
  foo X → false
```

This declaration generates a rule for the type-indexed function `foo` whose first argument is the type witness `(#list X`$_A$`)`, the result of the testify function for the type *list* A of the instance declaration:

```
foo (#list X_A) X → false
```

To translate a rule, we translate all its components. Notice that according to our source syntax, patterns $\overline{t}$ do not contain overloaded

function symbols, so they are decorated with types with empty contexts $\langle\rangle$. Therefore type witnesses will not be added to patterns, and the translation function $trans_{expr}$ will only erase the type decorations. The most important case of $trans_{expr}$ is the translation of a function symbol. When we have an overloaded function, we have to provide the type witnesses it needs. In this case we inspect the saturated context ϕ, collecting those types constrained by a marked class name and adding their associated type witnesses. The order in which these type witnesses are supplied is important, and must be the same for all the occurrences of the same overloaded function. Consider a possible occurrence of the previous function `f` applied to concrete types:

`f ::` $\langle eq^{\bullet}\ bool, ord\ bool, eq^{\bullet}\ (list\ int)\rangle \Rightarrow bool \rightarrow (list\ int) \rightarrow bool$

The translation of this decorated function symbol adds type witnesses for booleans and lists of integers, which are the types constrained by marked class names in the context:

```
f #bool (#list #int)
```

Notice that in expressions not containing overloaded functions, the result of the translation is the original expression without type decorations in functions symbols. The same happens with programs no containing overloaded functions. Therefore in these cases the translation does not introduce any overhead in the program.

As the reader can notice, the translation does not need the complete decoration of function symbols but only the types marked with a $^{\bullet}$ in the context. We have decided to use the complete inferred decorations to make more notable the close link between the translation and the type checking phase.

3.3 Important issues for the translation

The type checking phase is very important for this translation, since the information it provides in the contexts of the types that decorates function symbols directs the translation. There are two important issues that the type checker must address: context reduction and the marking of class names in contexts.

Context reduction

When performing the type checking of functions, the type checker infers a type τ and a context of class constraints. Consider the nondeterministic function `f`, where `gt` is the *greater* function with type $\langle ord\ A\rangle \Rightarrow A \rightarrow A \rightarrow bool$ and `eq` the equality function with type $\langle eq\ A\rangle \Rightarrow A \rightarrow A \rightarrow bool$:

```
f (X:Xs) Z → gt X Z
f (X:Xs) Z → and (eq X Z) (eq Xs [Z])
```

For these rules, the inferred type is $(list\ A) \rightarrow A \rightarrow bool$ and the context is $\langle ord\ A, eq\ A, eq\ (list\ A)\rangle$. The constraint $ord\ A$ comes from the order comparison in the first rule `gt X Z`, the constraint $eq\ A$ from the equality comparison between Z and the head of the list X, and the constraint $eq\ (list\ A)$ from the equality comparison `eq Xs [Z]`. However, this context contains some redundant information and could be reduced. There are three rules for context reduction:

- *Eliminating duplicate constraints*. We can reduce the context $\langle eq\ A, eq\ A\rangle$ to $\langle eq\ A\rangle$ and no information is lost.
- *Using instance declarations*. The usual instance declaration for equality on lists is `instance eq A ⇒ eq (list A) where (...)`, specifying how to use the equality on values A to define an equality on $list\ A$. Therefore, we can reduce the context $\langle eq\ A, eq\ (list\ A)\rangle$ to $\langle eq\ A\rangle$. This reduction is not a problem from the point of view of type witnesses, because given a type witness for A we can generate a type witness for $list\ A$.
- *Using class declarations*. The class declaration for `ord` is `class eq A ⇒ ord A where (...)`, specifying that any instance of `ord` is also an instance of `eq`. Therefore we can reduce the context $\langle ord\ A, eq\ A\rangle$ to $\langle ord\ A\rangle$. From the point of view of type witnesses this is not a problem, because we still know that we need a type witness of A.

Therefore, the previous context for function `f` would be reduced to $\langle ord\ A\rangle$ using all the previous rules. In [17] they explore different choices about how much context reduction to apply. Haskell's choice is to reduce the context completely before generalization, and this choice is necessary in our translation. Otherwise, the translation could generate rules that violate the restriction of linear left-hand sides. Consider the instance declaration for equality on pairs `instance` $\langle eq\ A, eq\ B\rangle \Rightarrow$ `eq (pair A B) where (...)`, and the rule `g P1 P2 →  ([fst P1, snd P2], eq P1 P2)`—where `fst` and `snd` project the first and second component of a pair respectively. If we do not use the instance declaration to reduce the context, the type decoration obtained for `g` is $\langle eq^{\bullet}\ (pair\ A\ A)\rangle \Rightarrow (pair\ A\ A) \rightarrow (pair\ A\ A) \rightarrow (pair\ (list\ A)\ bool)$. Then the left-hand side of the translated rule would be `g (#pair` X_A X_A`) P1 P2`. This is not syntactically valid in our target language as the data variable X_A appears twice. Applying two steps of context reduction using the instance and eliminating duplicates we obtain $\langle eq\ A\rangle$. With this new context the left-hand side of the translated rule is `g` X_A `P1 P2`, which now is valid in the target language.

Marking of class names

We have used marked class names in contexts to know which type witness to pass to functions. The task of marking class names is an easy task that must be done after type checking, when the types of all the functions are inferred. At this point, contexts will have only constraints on type variables due to context reduction. There can be more than one class constraint over the same type variable, however we do not want to pass duplicate type witnesses for the same type. That is the reason why we mark with a $^{\bullet}$ only one constraint per type variable, defining the order in which type witnesses must be passed. Consider a Fibonacci function that accepts any numeric argument and returns an integer:

```
fib N = if N<2 then 1 else fib (N-1) + fib (N-2)
```

Its inferred type is $\langle num\ A, ord\ A\rangle \Rightarrow A \rightarrow int$. However, we do not need to pass two identical type witnesses to the rule. Therefore we mark one of the constraints over A, obtaining the type $\langle num^{\bullet}\ A, ord\ A\rangle \Rightarrow A \rightarrow int$. Then in every call of the `fib` function we will only pass one type witness. Moreover, if we do not use the $^{\bullet}$ marks the left-hand side of the `fib` rule would be translated into `fib` X_A X_A `N`, with two occurrences of the data variable X_A, violating the syntactic constraint that patterns in a left-hand side of a rule are linear.

3.4 Case study: equality and order

Fig. 5 contains the translation of a complete program using equality and order. Fig. 5-a) shows the source program with type declarations in the function symbols. These decorations are introduced by the type checker so the user does not need to write them in the source program. We suppose that usual booleans functions `and, or`$::\langle\rangle \Rightarrow bool \rightarrow bool \rightarrow bool$ and the conditional function `ifthen`$::\langle\rangle \Rightarrow bool \rightarrow A \rightarrow A \rightarrow bool$ are defined. We also assume that functions for equality and ordering are defined for booleans and integers: `eqBool`, `eqInt`, `gtBool` and `gtInt`. Notice that the type checker has marked with a $^{\bullet}$ the classes *eq* and *ord* in the types of `eq` and `gt` respectively, as can be seen in the decorations of the different occurrences of these functions. We have defined the `eq` and `gt` functions for booleans and integers using two variables `X` and `Y` as arguments so that the rules have arity 2, instead of defining them as `eq = eqBool`, `eq = eqInt`, etc. The reason for this is that because of HO-patterns, we need that all the rules for overloaded functions have the same arity, as we will dis-

```
class eq A where
  eq :: A → A → bool

instance eq bool where
  eq X Y = eqBool::⟨⟩ ⇒ bool → bool → bool X Y

instance eq int where
  eq X Y = eqInt::⟨⟩ ⇒ int → int → bool X Y

instance ⟨eq A, eq B⟩ ⇒ eq (pair A B) where
  eq (U,V) (X,Y) = and::⟨⟩ ⇒ bool → bool → bool
    (eq::⟨eq• A⟩ ⇒ A → A → bool U X)
    (eq::⟨eq• B⟩ ⇒ B → B → bool V Y)

instance ⟨eq A⟩ ⇒ eq (list A) where
  eq [] [] = true
  eq [] (Y:Ys) = false
  eq (X:Xs) [] = false
  eq (X:Xs) (Y:Ys) = and::⟨⟩ ⇒ bool → bool → bool
    (eq::⟨eq• A⟩ ⇒ A → A → bool X Y)
    (eq::⟨eq• (list A)⟩ ⇒ (list A) → (list A) → bool Xs Ys)

member :: ⟨eq• A⟩ ⇒ (list A) → A → bool
member::⟨eq• A⟩ ⇒ (list A) → A → bool [] Y = false
member::⟨eq• A⟩ ⇒ (list A) → A → bool (X:Xs) Y =
    or::⟨⟩ ⇒ bool → bool → bool
    (eq::⟨eq•A⟩ ⇒ A → A → bool X Y)
    (member::⟨eq• A⟩ ⇒ (list A) → A → bool Xs Y)

class ⟨eq A⟩ ⇒ ord A where
  gt :: A → A → bool

instance ord bool where
  gt X Y = gtBool::⟨⟩ ⇒ bool → bool → bool X Y

instance ord int where
  gt X Y = gtInt::⟨⟩ ⇒ int → int → bool X Y

memberOrd :: ⟨ord• A⟩ ⇒ (list A) → A → bool
memberOrd::⟨ord• A⟩ ⇒ (list A) → A → bool [] Y = false
memberOrd::⟨ord• A⟩ ⇒ (list A) → A → bool (X:Xs) Y = ifthen
  (gt::⟨ord• A⟩ ⇒ A → A → bool X Y) false
memberOrd::⟨ord• A⟩ ⇒ (list A) → A → bool (X:Xs) Y = ifthen
  (eq::⟨eq• A⟩ ⇒ A → A → bool X Y) true
memberOrd::⟨ord• A⟩ ⇒ (list A) → A → bool (X:Xs) Y = ifthen
  (gt::⟨ord• A⟩ ⇒ A → A → bool Y X)
  (memberOrd::⟨ord• A⟩ ⇒ (list A) → A → bool Xs Y)
```

a) Source program with type decorations

```
eq :: A → A → A → bool
eq #bool X Y = eqBool X Y
eq #int X Y = eqInt X Y
eq (#pair X_A X_B) (U,V) (X,Y) = and
  (eq X_A U X)
  (eq X_B V Y)
eq (#list X_A) [] [] = true
eq (#list X_A) [] (Y:Ys) = false
eq (#list X_A) (X:Xs) [] = false
eq (#list X_A) (X:Xs) (Y:Ys) = and
  (eq X_A X Y)
  (eq (#list X_A) Xs Ys)

member :: A → (list A) → A → bool
member X_A [] Y = false
member X_A (X:Xs) Y = or
    (eq X_A X Y)
    (member X_A Xs Y)

gt :: A → A → A → bool
gt #bool X Y = gtBool X Y
gt #int X Y = gtInt X Y

memberOrd :: A → (list A) → A → bool
memberOrd X_A [] Y = false
memberOrd X_A (X:Xs) Y = ifthen
  (gt X_A X Y) false
memberOrd X_A (X:Xs) Y = ifthen
  (eq X_A X Y) true
memberOrd X_A (X:Xs) Y = ifthen
  (gt X_A Y X)
  (memberOrd X_A Xs Y)
```

b) Translated program

Figure 5. Translation of a program using equality and order

cuss in Sect. 5.3. Notice how the type checker decorates function symbols with the corresponding type instantiated to the concrete type used in the application. This is the case of the second occurrence of `eq` in the last rule of the instance *eq (list A)*, which has the decoration $\langle eq^\bullet\ (list\ A)\rangle \Rightarrow (list\ A) \to (list\ A) \to bool$ since `eq` is applied to lists. Fig. 5-b) shows the result of applying the translation of Def. 3 to the source program. Notice how the same type variable A in the decorations generates the same data variable X_A in the translated program—see for example the second rule for `member`. This is important since all these occurrences represent the same type witness that is passed as an argument.

4. Advantages of the Translation

In this section we show some of the benefits of the proposed translation compared to the classical dictionary-based one in FLP.

4.1 Efficiency

To test the efficiency of the proposed translation against the classical translation using dictionaries [10, 30], we have elaborated 7 different programs using type classes. We have chosen programs that can be part of real functional-logic programs and use the standard type classes *eq, ord* and *num*:

- *eqlist*: equality comparison between lists of integers.
- *fib*: Fibonacci function that accepts numeric arguments.
- *galeprimes*: sieve of prime numbers using a function of difference of sorted lists.
- *memberord*: member function in sorted lists.
- *mergesort*: John von Neumann's sorting algorithm.
- *permutsort*: sorting by selecting a sorted permutation of the original list.
- *quicksort*: C.A.R. Hoare's sorting algorithm.

The programs *fib, galeprimes, mergesort* and *quicksort* have been adapted from the suite of benchmark programs for Haskell implementations *nobench* [28]. Although *permutsort* is an inefficient sorting algorithm, we have included it in the set of tests because it is an example of the *generate-and-test* scheme, a kind of programs combining non-determinism and lazy evaluation, for which FLP obtains better results than functional or logic programs [8].

Program	Speedup	Speedup (Optimized)
eqlist	1.6414	1.3627
fib	2.3063	2.3777
galeprimes	1.4885	1.0016
memberord	2.2802	2.2386
mergesort	1.0476	1.0453
permutsort	1.7186	1.7259
quicksort	1.0743	1.0005

Figure 6. Speedup of the proposed translation over the classical translation using dictionaries

For each program we have measured in Toy the elapsed time in the evaluation of 100 random expressions in both translations. Translated programs using dictionaries are valid programs in Toy since it has a Damas-Milner type system. However, Toy has not integrated the liberal type system for FLP presented in [20]. In order to compile and execute the translated programs with type-indexed functions and type witnesses—which are not correct with respect to a Damas-Milner type system—we have used a especial version of Toy without the type checking phase. This does not distort the measures since once compiled Toy programs do not carry any type information at run time, so compiled programs are the same regardless the type system. For each expression we have calculated the speedup: the elapsed time in the translated program with dictionaries divided by the elapsed time in the translated program using type-indexed functions and type witnesses, and we have computed the mean speedup of the 100 tests. The results appear in the second column of Fig. 6. The biggest speedups are obtained in *fib* and *memberord*. The reason for the speed gain in *fib* is that the function `fib` needs two dictionaries—*ord* and *num*—but only one type witness, which means one extra matching each time `fib` is called. In *memberord* the reason is that it uses the overloaded function `eq` with every element. This function is contained in the *eq* dictionary which is inside the *ord* dictionary, so before apply it we have to extract the *eq* dictionary. This projection is not needed with type witnesses. The programs *permutsort*, *eqlist* and *galeprimes* also obtain a good speedup. In the case of *eqlist*, the reason of the speedup is that the `eq` function builds the dictionary of equality on lists in each recursive call. However, the same type witness argument for lists is passed to the recursive call. The rest of programs—*mergesort* and *quicksort*—do not obtain any improvement and run as fast as with dictionaries.

There are some well-known optimizations that can be applied to the translation using dictionaries [4, 11]. However, in the translation using type-indexed functions and type witnesses there is also room for optimizations. Therefore we have measured the speedup of the same programs when optimizations are applied to both translations. For the dictionary-based translation we have considered those optimizations from [4] applicable to our set of tests. For each test program, the following optimizations have been applied in sequence:

- *Flattening of dictionaries*: expand class dictionaries to contain both the methods of the class and all its superclasses. The dictionary of the superclasses is kept as well as flatting it, because it is sometimes needed.
- *Constant folding*: eliminate the method projection from a dictionary when the concrete dictionary is known. For example, `arb dictArbBool` is replaced by `arbBool`—see Fig. 1-b).
- *Automatic function specialization*: generate an specialized version of a function when it is applied to a concrete dictionary. This optimization has been only applied to `galeprimes`, since it is the only tested program whose code contains a function that is applied to a concrete dictionary.

The rest of optimizations presented in [4] have not been considered because they are dependent on the underlying architecture, which is different between Haskell and Toy, or because they address specific problems which do not appear in our test programs—as programming with complex numbers.

For the proposed translation using type-indexed functions and type witnesses the considered optimizations are:

- *Specialized version from instances*: Apart from the generated rules for the type-indexed functions, instances also generate specialized versions of the overloaded functions. For example, the instance `instance` $\langle eq\ \ A\rangle \Rightarrow eq\ \ (list\ \ A)$ from Fig. 5-a) generates the function `eq_list`:

  ```
  eq_list :: A → (list A) → (list A) → bool
  eq_list X_A [] [] = true
  eq_list X_A [] (Y:Ys) = false
  eq_list X_A (X:Xs) [] = false
  eq_list X_A (X:Xs) (Y:Ys) =
     and (eq X_A X Y) (eq_list X_A Xs Ys)
  ```

  Any occurrence of an overloaded symbol applied to a concrete type witness is replaced by the specialized version: `eq (#list bool)` is replaced by `eq_list #bool`, `ord #nat` by `ord_nat`, etc.
- *Automatic function specialization*: The same optimization explained before, but used when a function is applied to a concrete type witness. This optimization has been only applied to `galeprimes` for the same reasons as before.

The speedup results of the optimized versions appear in the third column of Fig. 6. For the programs *fib*, *memberord*, *mergesort*, *permutsort* and *quicksort*, the speedup does not change substantially. The reason is that dictionary optimizations do not affect the target program—with the exception of a constant folding in the definition of the `ord` dictionaries that is used once per test—and the specialized version of the type-indexed functions are not used. For the program *eqlist* the optimizations avoid the creation of the equality dictionary for lists—in the dictionary-based translation—and make use of the specialized version of equality for list—in the type-passing translation. The speedup decreases but the program with type-indexed functions and type witnesses still runs faster. For the *galeprimes* program there is no speedup since after applying the optimization to both translations the resulting code is similar because of the automatic function specialization.

The code of the tested programs and detailed results of the tests can be found in `http://gpd.sip.ucm.es/enrique/publications/pepm11/testPrograms.zip`.

4.2 Adequacy to call-time choice

Apart from the improvement in efficiency, the proposed translation also solves the problem of missing answers when combining non-determinism and overloading presented in Sect. 1. The problem is that dictionaries are shared, and non-deterministic nullary member functions inside them are evaluated to the same value in all the copies. With the proposed translation this problem does not arise because member function are not projecting functions that extracts from dictionaries but type-indexed functions that accepts a type witness as an argument. This type witness is shared as dictionaries, but each occurrence of the member function is a different application so they can generate different values.

The translation using type-indexed functions and type witnesses of the program containing the `arb` class appeared in Fig. 1-a) is:

```
arb :: A → A
arb #bool → false
```

```
arb #bool → true

arbL2 :: A → (list A)
arbL2 X_A → [arb X_A, arb X_A]
```

The class and instance declaration have generated the type-indexed `arb` function with two rules for booleans, and `arbL2` is translated to accept a type witness and pass it to the `arb` functions in its right-hand side. In this case the translation of the expression `arbL2::(list bool)` is `arbL2 #bool`, which can be reduced to `[arb #bool, arb #bool]` using the rule for `arbL2`. Here the first occurrence of `arb #bool` in the list can be reduced to `false` and the second to `true` using the different rules for `arb`, so it produces the answer `[false, true]` that was missing. In a similar way `arbL2 #bool` can be reduced to `[true, false]`.

The problem with non-deterministic nullary member functions and the dictionary-based translation could be solved if they are automatically replaced by functions of arity 1. This way, dictionaries do not contain functions that can be evaluated but HO-patterns—functions partially applied—that are values and can be shared without problem. However this solution presents some problems that are further discussed in Sect. 5.2.

4.3 Simplicity

From the point of view of difficulty, both translations—the dictionary-based and the proposed one—have a similar complexity: a type checking phase and a translation that uses the obtained type information. However, translated programs using the proposed translation are simpler than those obtained using the dictionary-based one. They are shorter, since they declare less data types and functions. Besides, type witnesses are first-order data, unlike dictionaries which are higher-order data containing functions. Finally, type witnesses have in most cases a simpler structure and are smaller than dictionaries.

With the two translations, obtained programs are the result of an automatic procedure integrated in the compiler, so the simplicity of obtained programs is not so important from the point of view of the user. However, it might be useful for later analyses or manipulations of translated programs. Furthermore, as we have seen in Sect. 4.1 and Sect. 4.2, this simplicity comes with an improvement of the efficiency and a better adequacy to call-time choice.

5. Discussion

In this section we discuss some additional aspects, including some problems, that arise with the translations of type classes in FLP.

5.1 Multiple modules and separate compilation

The dictionary-based translation combines well with multiple modules and separate compilation. A class declaration defines a datatype and some projecting functions, and instances define concrete values of the dictionary type. Therefore different instances can be compiled separately and joined later. With the proposed translation using type-indexed functions and type witnesses this seems more difficult. The problem is that generated type-indexed functions are *open* functions [18]: there is *one* type-indexed function per member function, but the rules can be spread in several modules. However, this is not a problem in Toy due to its code generation method and the demand of the type-indexed functions generated from member functions of classes. Toy programs use a demand driven strategy [19] for evaluating function applications. Consider a `leq` function on Peano natural numbers defined as:

```
leq z Y         = true
leq (s X) z     = false
leq (s X) (s Y) = leq X Y
```

In this case, the first argument is demanded in all the rules, and the second argument is demanded only in the second and third rules. Then the strategy is to evaluate the first argument to *head-normal form*. If it is the constructor `z`, then we apply the first rule. If it is the constructor `s` we evaluate the second argument of the rule. If the evaluation of that argument is the constructor `z` we apply the second rule. Otherwise if it is the constructor `s` we apply the third rule. The Prolog code generated for this function is[3]:

```
leq(A,B,H) :- hnf(A,HA), leq_1(HA,B,H).

leq_1(z,B,true).
leq_1(s(X),B,H) :- hnf(B,HB), leq_1_2(s(X),HB,H).

leq_1_2(s(X),z,false).
leq_1_2(s(X),s(Y),H) :- leq(X,Y,H).
```

The predicate `hnf` is a built-in predicate that computes *head normal forms*. The predicate `leq` is the main predicate to evaluate the `leq` function. It uses the predicates `leq_1` and `leq_1_2`, where the numbers represent in which positions a head normal form has been previously obtained. Notice that the last argument of the predicates represents the result. It is easy to see that these predicates follow the demand driven strategy explained before.

The peculiarity of translated member functions is that they always have a constructor in their first argument: the type-witness. Therefore their first argument is always demanded in all the rules translated from the instances, so the strategy is to evaluate it to head normal form. Consider the `eq` function in Fig. 5-b). Since the first argument is demanded in all the rules, we generate the predicate to evaluate the type witness to head normal form:

```
eq(W,A,B,H) :- hnf(W,HW), eq_1(HW,A,B,H).
```

We also generate the predicate `eq_1` with clauses for the different instances:

```
eq_1(#bool,A,B,H) :- eqBool(A,B,H).
eq_1(#int,A,B,H) :- eqInt(A,B,H).
eq_1(#pair(WA,WB),A,B,H) :- (...)
eq_1(#list(WA),A,B,H) :- (...)
```

If each instance of `eq` is in a different module, we compile them separately. However, in each translated module the first argument of `eq` is uniformly demanded, so we generate the predicate `eq/4` as before and the corresponding clauses for `eq_1/4` and the rest of predicates. Notice that in the translated rules for equality on pairs and list, the three arguments are uniformly demanded. In these cases we chose from left to right, so we always generate the same clause for `eq/4` that computes the head-normal form of the first argument and calls to `eq_1/4`. In the compilation of a program that imports the different modules with the instances, the code for the `eq` function is obtained by simply joining the predicates `eq/4`, `eq_1/4` ... from the compiled modules. Each compiled module contains a clause for `eq/4`, so it is important to remove those duplicates in the final compiled program.

Notice that this solution is not valid for arbitrary open functions, since the demand of the arguments is unknown and the code generation would require an analysis with the rules from all the modules.

5.2 Possible solution for non-deterministic nullary member functions in the dictionary-based translation

The loss of expected answers that arises in the dictionary-based translation when non-deterministic nullary member functions are used could be solved if they are automatically replaced by unary functions. Fig. 7 shows the program translated with dictionaries from Fig. 1-a) where `arb` has been extended to an unary function accepting unit as argument. The translation of `arbL2::(list bool)` is `arbL2 dictArbBool` as in the original case, but now it reduces to `[arb dictArbBool (), arb dictArbBool ()]`. Although both copies of the dictionary are shared, now they can

[3] This is not the exact code generated by the Toy compiler. We have simplified it for the sake of conciseness.

```
data dictArb A = dictArb (unit → A)

arb :: dictArb A -> (unit → A)
arb (dictArb F) → F

arbBool :: unit → bool
arbBool () → false
arbBool () → true

dictArbBool :: dictArb bool
dictArbBool → dictArb arbBool

arbL2 :: dictArb A -> list A
arbL2 DA → [arb DA (), arb DA ()]
```

Figure 7. Translation of the program in Fig. 1-a) extending `arb` to have one argument

only be reduced to `dictArb arbBool`. It is now a value—notice that `arbBool` is a HO-pattern—so it cannot be reduced further. After the extraction of the `arbBool` function from the dictionary the expression is `[arbBool (), arbBool ()]`, which can be reduced to `[false, true]` or `[true, false]` applying the rule for `arbBool` for twice.

Since being non-deterministic is a typically undecidable property, the technique of adding the unit argument should be applied to every nullary member function, even if it is indeed deterministic. This will introduce an unnecessary overhead—apart from the inevitable overhead caused by dictionaries—to nullary deterministic member functions. We could consider an analysis to detect (in some cases) if the definition of a nullary member function in a concrete instance is deterministic. In those cases the extra unit argument could in principle be avoided. However this solution makes difficult separate compilation. The reason is that a later inclusion of a new module with an instance where the considered nullary member function is non-deterministic will force the recompilation of all the related modules: it will be necessary to change the dictionary declaration—now it contains a member function whose first argument is of type *unit*—and add the unit argument to the rules in the previous instances.

The translation using type-indexed functions and type witnesses proposed in this paper treats non-deterministic nullary member functions and the rest of member functions in a homogeneous way. Furthermore, it does not require recompilation and it does not add any extra overhead to deterministic nullary member functions—apart from the type-witness. Therefore, we believe that the proposed translation is a better option than the dictionary-based translation when dealing with the combination of non-determinism and nullary member functions.

5.3 Problems with arities and HO-patterns

In our FLP setting the arity of function symbols plays an important role to identify whether a function application forms a HO-pattern or it is totally applied and can be reduced. Therefore all the rules of the same function must have the same arity, and this property must be ensured in the target program. In FP the compiler checks that all the rules of a function have the same number of arguments, but this is not checked for the rules of member functions in different instances. However, this property must be checked if the proposed translation is used. The reason is that the rules of the same member function in different instances are translated to be the rules of the same type-indexed function. If the original rules from the instances have different arities, then the rules for the type-indexed functions will have different arities and the translated program will not be a valid FL program. To solve this problem we propose to annotate the arity of member functions in the class declaration. For example the class declaration for `eq` in Fig. 5-a) is changed to:

`class` $eq\ A$ `where`
  `eq/2 ::` $A \to A \to bool$

Using this arity declaration the compiler will be able to check if all the rules for `eq` have the same arity even if they belong to instances in different modules. Notice that this problem with arities does not appear in the dictionary-based translation since the rules of a member function in an instance generates a specialized function—see `arbBool` in Fig. 1-b)—and the member function itself is transformed into a function which projects from the dictionary.

Another problem to address is the occurrence of HO-patterns containing overloaded functions in the patterns of the left-hand side of rules. If this kind of functions appear in the patterns, the type checking stage will decorate them with an overloaded type. Besides, class constraints coming from the overloaded function could remain after context reduction, so the defined function symbol will have an overloaded type containing them. In this situation the proposed translation will generate non-linear functions. Consider the program from Fig. 5-a) and the rule that uses the HO-pattern `eq`:

`f eq` $\to$ `true`

After the type checking stage the rule is decorated as:

`f::`$\langle eq^{\bullet}\ A\rangle \Rightarrow (A \to A \to bool) \to bool$
  `eq::`$\langle eq^{\bullet}\ A\rangle \Rightarrow A \to A \to bool \quad \to \quad$ `true`

so the translated rule would be:

`f` X_A `(eq` X_A`)` $\to$ `true`

This rule is invalid in our setting, since the variable X_A appears twice in the left-hand side so the patterns are non-linear. Notice that this problem also appears in the dictionary-based translation since the same variable representing the dictionary would be passed as the extra argument of `f` and `eq`.

A possible solution to this problem might be not to translate the patterns in the left-hand sides of the rules, so no type witnesses would be added to the overloaded functions in patterns. Since the class constraints from these functions remain in the context of the defined function, they will generate the type witnesses as the first arguments of the defined function. However, this solution leads to a loss of expected answers. Consider the same function rule for `f`. If we do not translate the patterns, the translated rule would be:

`f` X_A `eq` $\to$ `true`

which now is linear. The value `true` is an expected answer of the evaluation of `f eq::`$bool \to bool \to bool$—we have added the type decoration to `eq` to avoid ambiguity. The type checker would extend this expression with complete type decorations:

`f::`$\langle eq^{\bullet}\ bool\rangle \Rightarrow (bool \to bool \to bool) \to bool$
  `eq::`$\langle eq^{\bullet}\ bool\rangle \Rightarrow bool \to bool \to bool$

and the translation of this expression would be:

```
f #bool (eq #bool)
```

However this translated expression does not match with the head of the rule `f` X_A `eq`, so it cannot be reduced to `true`. Notice that it also happens with the dictionary-based translation. The translation of the rule would be the same, as `f` needs an extra argument containing the dictionary of equality. The translation of the expression would add two dictionaries for the equality on booleans:

```
f dictEqBool (eq dicEqBool)
```

This translated expression cannot be reduced to the value `true` either. It does not match with the head of the rule for `f`, but the subexpression `eq dicEqBool` can be reduced to `eqBool`—assuming that `eqBool` is the function inside the dictionary of equality for booleans. However the resulting expression `f dictEqBool eqBool` cannot be reduced to `true` using the rule `f` X_A `eq` $\to$ `true` because it does not match with its head.

Considering the problems that HO-patterns containing overloaded functions in the left-hand side of rules cause in both translations, it seems a good design choice to prohibit the occurrence of

overloaded functions in the patterns in the left-hand side of rules. However HO-patterns are a very expressive feature of FLP, so this problem must be further studied in order to find a solution.

6. Concluding Remarks and Future Work

In this paper we have proposed a translation for type classes in FLP following a type-passing scheme [29]. The translation uses type-indexed functions and type witnesses, and translated programs are well-typed wrt. a new liberal type system for FLP [20]. We argue that the proposed translation is a good design choice to implement type classes in FLP because it improves on the standard dictionary-based translation in some points:

- Our tests show that translated programs using type-indexed functions and type witnesses perform faster—in general—than those using the dictionary-based translation [10, 30]. The tests also show that if we apply optimizations to both translated programs, those using type-indexed functions and type witnesses still perform faster, although the difference in this case is smaller.
- It does not present the problem of missing answers which appears with the dictionary-based translation in programs that use non-deterministic nullary member functions [24].
- The proposed translation consists in simple steps that make use of type decorations for function symbols obtained by usual type checking algorithms supporting type classes [5, 27], so it does not add extra complications over the standard dictionary-based translation. Besides, translated programs using the proposed translation are shorter and simpler than those generated using the dictionary-based translation.
- Although it needs some special treatment, the proposed translation supports multiple modules and separate compilation in an easy way.

We consider some lines of future work. The first is the implementation of the complete translation into the Toy system. Since the translation rules are pretty simple, the hard step is implementing the standard type checker supporting type classes and place the type decorations in the function symbols. Once the translation is implemented, we will be able to test the efficiency results with a larger set of programs. We also want to study if the proposed translation supports easily well-known extensions of type classes like *multi-parameter type classes* [17] or *constructor classes* [16] for FLP. According to [29], these extensions fit easily in a type-passing translation scheme. Finally, we intend to study in further detail the problematic of HO-patterns using overloaded functions in the left-hand sides of rules, so that we can find better solutions than prohibit them.

Acknowledgments

This work has been partially supported by the Spanish projects TIN2008-06622-C03-01, S2009TIC-1465, UCM-BSCH-GR58/08-910502. We also want to acknowledge to Francisco López-Fraguas and Juan Rodríguez-Hortalá for their useful comments and ideas.

References

[1] Münster Curry compiler. `http://danae.uni-muenster.de/~lux/curry/`.

[2] Sloth Curry compiler. `http://babel.ls.fi.upm.es/research/Sloth/`.

[3] Zinc compiler. `http://zinc-project.sourceforge.net/`.

[4] L. Augustsson. Implementing Haskell overloading. In Proc. *FPCA '93*, pages 65–73, 1993.

[5] S. Blott. Type inference and type classes. In *Proc. of the 1989 Glasgow FP Workshop*, pages 254–265, 1990.

[6] J. Cheney and R. Hinze. First-class phantom types. Technical Report TR2003-1901, Cornell University, July 2003.

[7] L. Damas and R. Milner. Principal type-schemes for functional programs. In Proc. *POPL '82*, pages 207–212, 1982.

[8] J. C. González-Moreno, M. T. Hortalá-González, F. J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.

[9] J. C. González-Moreno, M. T. Hortalá-González, and M. Rodríguez-Artalejo. Polymorphic types in functional logic programming. *Journal of Functional and Logic Programming*, 2001(1), July 2001.

[10] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[11] K. Hammond and S. Blott. Implementing Haskell type classes. In *Proc. of the 1989 Glasgow FP Workshop*, pages 266–286, 1990.

[12] M. Hanus. Multi-paradigm declarative languages. In Proc. *ICLP 2007*, volume 4670 of *LNCS*, pages 45–75. Springer, 2007.

[13] M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry/report.html`, March 2006.

[14] R. Hinze and A. Löh. Generic programming, now! In *Datatype-Generic Programming 2006*, volume 4719 of *LNCS*, pages 150–208. Springer, 2007.

[15] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In Proc. *HOPL III*, pages 12–1–12–55, 2007.

[16] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In Proc. *FPCA '93*, pages 52–61, 1993.

[17] S. P. Jones, M. Jones, and E. Meijer. Type classes: An exploration of the design space. In *Haskell Workshop*, 1997.

[18] A. Löh and R. Hinze. Open data types and open functions. In Proc. *PPDP '06*, pages 133–144, 2006.

[19] R. Loogen, F. J. López-Fraguas, and M. Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In Proc. *PLILP '93*, pages 184–200, 1993.

[20] F. J. López-Fraguas, E. Martin-Martin, and J. Rodríguez-Hortalá. Liberal Typing for Functional Logic Programs. *To appear APLAS 2010*. Available at `http://gpd.sip.ucm.es/enrique/publications/liberalTypingFLP/aplas2010.pdf`.

[21] F. J. López-Fraguas, E. Martin-Martin, and J. Rodríguez-Hortalá. New results on type systems for functional logic programming. Volume 5979 of *LNCS*, pages 128–144. Springer, 2010.

[22] F. J. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multi-paradigm declarative system. In Proc. *RTA'99*, volume 1631 of *LNCS*, pages 244–247. Springer, 1999.

[23] W. Lux. Adding Haskell-style overloading to Curry. In *Workshop of Working Group 2.1.4 of the German Computing Science Association GI*, pages 67–76, 2008.

[24] W. Lux. Type-classes and call-time choice vs. run-time choice - Post to the Curry mailing list. `http://www.informatik.uni-kiel.de/~curry/listarchive/0790.html`, 2009.

[25] J. C. González-Moreno, M. T. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In Proc. *ICLP'97*, pages 153–167, 1997.

[26] J. J. Moreno-Navarro, J. Mariño, A. del Pozo-Pietro, Á. Herranz-Nieva, and J. García-Martín. Adding type classes to functional-logic languages. In *1996 Joint Conf. on Declarative Programming, APPIA-GULP-PRODE'96*, pages 427–438, 1996.

[27] T. Nipkow and C. Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

[28] D. Stewart. nobench: Benchmarking Haskell implementations. `http://www.cse.unsw.edu.au/~dons/nobench.html`.

[29] S. R. Thatté. Semantics of type classes revisited. In Proc. *LFP '94*, pages 208–219, 1994.

[30] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In Proc. *POPL '89*, pages 60–76, 1989.

[31] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. *SIGPLAN Not.*, 38(1):224–235, 2003. ISSN 0362-1340.